# Primer Trabajo Práctico

14 de Abril de 2010 — Algoritmos y Estructuras de Datos III


| Integrante | LU | Correo electrónico |
|---|---|---|
| Bianchi, Mariano | 92/08 | bianchi-mariano@hotmail.com |
| Brusco, Pablo | 527/08 | pablo.brusco@gmail.com |
| Di Pietro, Carlos Augusto Lyon | 126/08 | cdipietro@dc.uba.ar |

**Facultad de Ciencias Exactas y Naturales**

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

# Índice

# 1. Ejercicio 1

## 1.1. Introducción

El primer problema del presente trabajo consistió en la implementación de un algoritmo capaz de dar solución a la ecuación

$$b^n \ mod \ (n) \tag{1}$$

haciendo uso de alguna de las técnicas algorítmicas aprendidas hasta el momento en la materia. Asímismo, la consigna dictaba que la complejidad final del algoritmo debería ser menor a $O(n)$.

En pos de cumplimentar lo pedido se decidió usar la técnica de *Dividir & Conquistar*[1] para desarrollar el algoritmo. Esta técnica se caracteriza principalmente en dividir la instancia de un problema en instancias más pequeñas, atacar cada una de ellas por separado y resolverlas, para finalmente juntar sus resultados y así producir el resultado final.

## 1.2. Explicación

La primera solución que se piensa casi de manera intuitiva es la de mutliplicar $n$ veces el número $b$ y luego hallar el resto de dividir ese resultado por $n$.

$$\underbrace{b.b.b \ldots b.b.b}_{n \ veces} \ mod \ (n) = b^n \ mod \ (n) \tag{2}$$

Sin embargo, la complejidad ese algoritmo es $O(n)$, ya que se realizan $n$ multipicaciones y 1 división, razón por lo cual no cumple con lo pedido en la consigna. Además, puesto que ese algoritmo realizar sucesivas multipliciones de $b$, cabe la posibilidad de que el valor en que se va acumulando el resultado sobrepase el máximo número entero representable en la computadora produciedo así un overflow y obteniéndose en consecuencia un resultado incorrecto.

Analizando la falla del algoritmo anterior, se observa que lo que se debe evitar es calcular directamente el resultado de $b^n$ ya que ese número podría ser excesivamente grande.
Con esto en mente veamos una manera más conveniente de abordar el problema. Es claro, por definición de congruencia, que resover la ecuación en (1) equivale a hallar el $x$ tal que:

$$b^n \ \equiv \ x \ (n) \ , \quad con \ 0 \ \leq \ x \ < \ n \tag{3}$$

Luego, asumamos por un momento que podemos tener gratis un k tal que:

$$b^{n/2} \ \equiv \ k \ (n) \ , \quad con \ 0 \ \leq \ k \ < \ n \tag{4}$$

Entonces, aplicando el resultado de (4) en (3) obtenemos que:

[1] Poner alguna referencia en donde se explique esta técnica

- Si $n$ es par:

$$\begin{aligned} b^n &\equiv x\ (n)\ , \quad con\ 0 \leq x < n \\ b^{n/2}.b^{n/2} &\equiv x\ (n) \\ k.k &\equiv x\ (n) \\ k^2 &\equiv x\ (n) \end{aligned} \tag{5}$$

- Si $n$ es impar par:

$$\begin{aligned} b^n &\equiv x\ (n)\ , \quad con\ 0 \leq x < n \\ b.b^{n/2}.b^{n/2} &\equiv x\ (n) \\ b.k.k &\equiv x\ (n) \\ b.k^2 &\equiv x\ (n) \end{aligned} \tag{6}$$

Sin lugar a dudas, las expresiones resultantes en ambos casos son fáciles de resolver ya que $k < n$; al tiempo que también son números mucho menores que $b^n$.

Volviendo un poco sobre nuestro pasos, se dijo anteriormente en (4) que podíamos asumir que obtener $k$ no tenía un costo computacional significativo. No obstante, esto no siempre es cierto puesto que para obtener $k$ es necesario calcular previamente $b^{n/2}$ el cual puede ser un número nada despreciable en lo que respecta a tamaño en memoria.
Pero esto no representa, dificultad alguna para calcular $b^{n/2}$ puesto que podemos aplicar repetidas veces el procedimiento descripto con anterioridad dividiendo el exponente de $b$ hasta que valga 1, y luego reagrupando los resultados y tomando módulo $n$ hasta obtener finalmente la solución del problema.

## 1.3. Análisis de la complejidad del algoritmo

A continuación se calculará la complejidad del algortimo *potenciacion* que es el que resuelve el problema que se planteo en (1). Dicho problema tiene por variables a $b \in \mathbb{N}$ y $n \in \mathbb{N}$, lo cual en términos matemáticos representa que $b$ y $n$ pueden tomar valores tan grandes como se desee puesto que $\mathbb{N}$ es un conjunto infinito. Esto se ve traducido en que los parámetros del algoritmo *potenciacion* sean tan grandes como uno desee.
Ahora bien, la consigna asegura que $b$ puede considerarse como un número acotado; pero aún así, $n$ puede seguir siendo tan grande como uno desee. Por este motivo, resulta lógico pensar que la suposición de que $n$ cabe perfectamente en una posicion fija de memoria no es correcta en lo absoluto. Entonces, para que el análisis de complejidad del algoritmo sea lo más correcto posible se debe evaluar la complejidad de las operaciónes que realiza el algoritmo en función del tamaño de símbolos binarios necesarios para representar el valor de n dentro del ordenador.
Lo expresado en el pensamiento anterior conlleva a que la medición de complejidad del algortimo se efectúe mediante el *modelo logarítmico* el cual precisamente mide la performance de un algoritmo a través de la suma de los costos de todas las operaciones en función del tamaño de las variables involucradas.

Sigue a continuación un pseudocódigo del algoritmo en cuestión en el que se detallan, para cada instrucción, su costo en el modelo logarítmico:

*potenciacion(base :* $\mathbb{N}$, *exp:* $\mathbb{N}$, *m:* $\mathbb{N}$*)*

```
Complejidad: T(n)
var res : ℤ
base ← base mod m ;                                    // O(log₂(base))
if (base == 0);                                        // O(log₂(base))
then
    return base
else
    if (exp == 1);                                     // O(log₂(exp))
    then
        return base
    else
        if (exp mod 2 == 0) ;                          // O(log₂(exp))
        then
            var temp : ℤ
            temp ← potenciacion(b, exp/2, m) ;         // T(exp/2)
            temp ← (temp*temp) ;                       // O(log₂(log₂(temp)²))
            temp ← temp mod m ;                        // O(log₂(log₂(temp) * log₂(m)))
            res ← temp ;                               // O(log₂(temp))
        else
            var temp : ℤ
            temp ← potenciacion(b, (exp-1)/2, m) ;     // T((exp-1)/2)
            temp ← (temp*temp) ;                       // O(log₂(log₂(temp)²))
            temp ← temp mod m ;                        // O(log₂(log₂(temp) * log₂(m)))
            temp ← temp * base ;                       // O(log₂(log₂(temp) * log₂(b)))
            temp ← temp mod m ;                        // O(log₂(log₂(temp) * log₂(m)))
            res ← temp ;                               // O(log₂(temp))
        end
    end
end
return res
```

## 1.4. Detalles de Implementación

Dentro de la carpeta *./ej1/*, se puede encontrar un archivo ejecutable *ejercicio_1* compilado para GNU-Linux, el cual resuelve el problema anteriormente descripto. Este programa se ejecuta por consola mediante el comando `./ejercicio_1`, y recibe como parámetro los archivos de entrada *".in"* a procesar. Puede recibir tantos nombres de archivo como se desee, pero en caso de no recibir ninguno, el programa procesará el archivo *Tp1Ej1.in* que se encuentra incluído dentro de la misma carpeta.
Una vez ejecutado, el algoritmo procesa la cola de archivos que recibió como parámertos de

a uno por vez generando para cada uno de ellos dos archivos:

- Un archivo *".out"* omónimo con la solucion a la ecuación (1) para cada par de naturales (b, n) contenidos en el archivo de entrada. Este archivo se guarda en la misma carpeta en la que se encuentra el ejecutable *ejercicio_1*.
- Un archivo omónimo con el sufijo *"_grafico.out"*, en el cual registra para cada n el número de operaciones realizadas por el algoritmo. Este archivo se guarda en la carpeta *./ej1/info graficos*, y tiene por objetivo facilitar la tarea de cargar los datos en el programa de análisis gráfico *QtiPlot*.

Por otra parte, en la misma carpeta, se puede hallar un archivo ejecutable *input_gen1* también compilado para GNU-Linux. Al correr este programa desde la consola mediante el comando `./input_gen1` se despliega un menú de opciones para generar distintos tipos de archivos *".in"* para ser resueltos por el ejecutable *ejercicio_1*. Una vez elegido el tipo de entrada a crear, el programa solicita que se ingrese un nombre de archivo y la cantidad de casos a generar. Acto seguido guarda el archivo generado en la carpeta *./ej1/*.

Asímismo, en la carpeta *./ej1/* hay un Makefile el cual permite recompilar los archivos ejecutables con tan solo ejecutar el comando `make` en la consola. Además, ejecutando el comando `make clean` se pueden eliminar los archivos ejecutables y todos los archivos de extensión *.out*.

Luego, en la carpeta *./ej1/sources* se encuentran los codigos fuentes de los ejecutables antes descriptos. Los mismo están escritos en lenguaje C++ y tienen comentadas las partes relevantes para simplificar la comprensión.

Finalmente, en *./ej1/* se hayan los archivos *.Tp1Ej1.in* y *Tp1Ej1.out* que vienen junto con en el enunciado de este Trabajo Práctico, y se haya también la carpeta *./ej1/test* la cual contiene los archivos *".in"* generados para probar el algortimo que resuelve (1) junto con sus correspondientes gráficos de cantidad de operaciones promedio vs. n.

## 2. Ejercicio 2

### 2.1. Introducción

El segundo problema consistió en la implementación de un algoritmo capaz de dar solución al problema que se plantea a continuación:

Decidir si un grupo de n personas pueden formar o no una ronda que cumpla con las siguientes restricciones:

- La ronda debe contener a todas las personas.
- Algunas personas son amigas y otras no.
- Cada alumna debe tomar de la mano a dos de sus amigas.

### 2.2. Explicación

Dado que para este problema no se conocen algoritmos buenos[2], se pensó en utilizar la solución por fuerza bruta, es decir, intentar todas las combinaciones hasta lograr determinar si hay o no solución, pero, agregando algunas mejoras.

Mejoras como evaluar, en el momento de cargar la ronda, si alguna persona no tiene las suficientes amigos, es decir 2, o si todos son amigos de todos.

Luego, se pensó en la estrategia de "Vuelta atrás", (Backtracking).

***Backtracking:***

*En su forma básica, la idea de backtracking se asemeja a un recorrido en profundidad dentro de un grafo dirigido. El grafo en cuestión suele ser un árbol, o por lo menos no contiene ciclos. Sea cual sea su estructura, existe sólo implícitamente. El objetivo del recorrido es encontrar soluciones para algún problema. Esto se consigue construyendo soluciones parciales a medida que progresa el recorrido; estas soluciones parciales limitan las regiones en las que se puede encontrar una solución completa. El recorrido tiene éxito si, procediendo de esta forma, se puede definir por completo una solución. En este caso el algoritmo puede bien detenerse (si lo único que se necesita es una solución del problema) o bien seguir buscando soluciones alternativas (si deseamos examinarlas todas). Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar. En tal caso, el recorrido vuelve atrás exactamente igual que en un recorrido en profundidad, eliminando sobre la marcha los elementos que se hubieran añadido en cada fase. Cuando vuelve a un nodo que tiene uno o más vecinos sin explorar, prosigue el recorrido de una solución.* [3]

En este caso particular, la idea es, seleccionar una persona (P), ingresarla en la ronda, e ir armando un arbol implicito de posibilidades en donde el nodo padre es P y cada rama se forma con las distintas selecciones de amigas de P y sus sub-arboles, con las combinaciones posibles de amigas del nodo padre en el que esté. Para ello se utiliza la recursión, en donde se

[2] Un algoritmo se considera bueno si puede ser resuelto en tiempo polinomial.

[3] http:es.wikipedia.org

intenta bajar lo más posible en el arbol hasta llegar a formar un camino desde el primer nodo hasta alguna hoja en donde se encuentren todas las chicas con nodos padre e hijo amigas.

También, se implementaron otras mejoras que intentan podar el arbol dadas ciertas condiciones que permitan acortar las posibilidades:

- Las amigas de un nodo se pueden procesar en cualquier orden, pero decidimos intentar ser lo más restrictivo posible con las primeras (las que tienen menos amigas primero). Este proceso poda el árbol de búsqueda antes de que se tome la decisión y se llame a la subrutina recursiva.
- Cuando se elige qué valor se va a asignar, se hace un examen comprobando que aun queden amigas de la que inicio la ronda dispoibles. En caso que no, se pueden evitar muchas llamadas sin sentido.

A continuación se ve un pseudocodigo que muestra el comportamiento de la estrategia y las mejoras elegidas.

Algunos detalles:

- **gente** es una lista con elementos de tipo chica.
- el tipo **chica** consta de un nombre (int) y sus amigas (conjunto de nombres).
- **enRonda** es un conjunto de nombres donde se va almacenando quienes pertenecen a la ronda.
- **amigasPrimera** es un conjunto de nombres donde se almacenan los nombres de las amigas de la chica que comienza la ronda.
- **tam(c)** devuelve el tamaño del conjunto c.
- **insertar(c,e)** inserta a e en el conjunto c.
- **borrar(c,e)** borra a e del conjunto c.
- **cuenta(c,e)** cuenta la cantidad de apariciones del elemento e en c.
- **primerElemento(l)** devuelve el primer elemento de una lista.

**resolver()**

```
Complejidad: T(n)
var n : int ← tam(gente)
var sonPocas : bool ← false
var sonSuficientes : bool ← false
foreach k en gente do
    sonPocas ← sonPocas or tam(amigas(k)) < 2;
    sonSuficientes = sonSuficientes && tam(amigas(k)) == n-1);
end
if (sonSuficientes) then
    retornar true
end
if (sonPocas) then
    retornar false
end
var solitaria : chica ← primerElemento(gente);
insertar(nombre(solitaria),enRonda);
retornar probarDistintasRondas(solitaria,solitaria);
```

**probarDistintasRondas(prim : chica, ult : chica)**

```
Complejidad: T(n)
if (cuenta(amigas(ult),nombre(prim)) ≠ 0 && tam(gente) = tam(enRonda)
) then
    retornar true;
else
    var res : bool
    var eraAmiga : bool ← false
    foreach chi en gente do
        var estaEnRonda : int ← cuenta(enRonda,nombre(chi))
        var esAmigaDeLaUltima : int ← cuenta(amigas(ult),nombre(chi))
        var restantesAmigasPrimera : int ← tam(amigasPrimera)
        var esAmigaDeLaPrimera : int ← cuenta(amigas(prim),nombre(chi))
        if ( (estaEnRonda = 0) && (esAmigaDeLaUltima ≠ 0) &&
        (restantesAmigasPrimera ≠ 0) ) then
            if (esAmigaDeLaPrimera ≠ 0) then
                eraAmiga ← true;
                eliminar(amigasPrimera,nobre(chi));
            else
                eraAmiga = false;
            end
            insertar(enRonda,nombre(chi));
            res ← probarDistintasRondas(prim,chi);
            if (res) then
                retornar true;
            else
                borrar(enRonda,nombre(chi));
                if (eraAmiga) then
                    insertar(amigasPrimera,nombre(chi))
                end
            end
        end
    end
    retornar false;
end
```

## 2.3. Análisis de la complejidad del algoritmo

La complejidad en el peor caso de este algoritmo surge inmediatamente al hacer pruebas. Dependiendo de las entradas elegidas, el programa podria tener que recorrer casi todas las ramas del arbol hasta poder decir que no hay solucion. En este caso, podriamos entonces solo acotar el peor caso por la funcion $f(n) = c.(n-1)! + b$ (donde b y c son constantes) ya que la cantidad de formas de armar una ronda con n personas es (n!/n) = (n-1)![4]

Por lo tanto, el algoritmo implementado NO es bueno. Sin embargo, cuando para un conjunto de datos de entrada se puede formar una ronda, el algoritmo no recorre todo el arbol en busca de la solución, sino que al encontrar una termina. Esta propiedad del backtracking y el resto de las mejoras, hace que para datos de entradas en donde se pueden formar rondas, el algoritmo se comporte como lo haria una solucion polinomial.

[4]Se lo puede imaginar como una fila de n personas. Para la primer posición hay n personas dispoibles, para la segunda n-1, y asi sucesivamente. Por lo tanto hay n*(n-1)*...*1 formas de acomodar las personas en una fila. Ahora, si se unen las puntas, podria rotar la fila n veces y seguiria siendo la misma ronda, por lo tanto se dividiendo por n obtengo la respuesta: **(n-1)!**.

# 3. Ejercicio 3

## 3.1. Introducción

El tercer y último problema de este trabajo práctico consistió en la implementación de un algoritmo que dé una solución al siguiente problema:
Dadas dos listas con horarios de entrada y salida de ciertos trabajadores a una empresa, decidir cuál es el mayor número de trabajadores que se encuentran al mismo tiempo dentro de dicha empresa. Estas listas cumplían con estas condiciones:

- Están ordenadas ascendentemente por horario.
- Las horas van desde 00:00:00 hasta 23:59:59
- Para todos los trabajadores, es cierto que su horario de ingreso es estrictamente menor que su horario de egreso.

## 3.2. Explicación

### 3.2.1. Primer aproximación

En una primer mirada, se pensó que la mejor forma de resolver el ejercicio era aplicando un algoritmo similar al del quicksort[5], sólo que el mismo se realizaba sobre conjuntos y no sobre arreglos. En resumidas cuentas, la idea era tomar un trabajador cualquiera del grupo, y separar en tres conjuntos distintos: los que se cruzaban con el trabajador pivote, los que salían antes que el pivote entrara y los que entraban luego que el pivote saliera[6]. De esta manera, si se repetía el proceso varias veces, se llegaba a tener varios conjuntos en los cuales aparecían solamente trabajadores que se cruzaban en sus horarios dentro de la empresa. Finalmente, sólo restaba devolver el mayor cardinal de dichos conjuntos.

Al igual que el quicksort, si el pivote era elegido al azar, el algoritmo anterior contaba con una complejidad promedio de $O(n * log(n))$ (donde $n$ es la cantidad de trabajadores). Igualmente, el peor caso seguía siendo $O(n^2)$, por lo que no se iba a poder respetar la cota dada como máxima para este trabajo, ya que se especificaba que el algoritmo debía tener una complejidad menor a $O(n^2)$.

Pero luego de un mejor análisis del problema, se cayó en la cuenta que los datos de entrada contaban con la característica de estar ordenados por horario (tanto los de entrada como los de salida), por lo que inmediatamente surgió la idea de poder solucionar el problema con una complejidad de $O(n)$ sin necesidad siquiera de una técnica algorítmica compleja.

### 3.2.2. Segunda aproximación

Para encontrar solución al problema dado, lo primero que se realizó fue pensar qué cosas se necesitaban para representarlo. Para ello se creó una clase "Empresa", la cual consta de

[5] Alguna referencia que explique el algoritmo

[6] Para que el algoritmo funcionara bien, se debía tener cierto cuidado con los trabajadores que se cruzaran con el pivote, pero la explicación de estos detalles no hacen a la esencia de la introducción. Además, este algoritmo fue descartado más adelante, por lo que estos detalles son irrelevantes para este trabajo.

dos arreglos que contienen en cada posición, una hora junto con un nombre de un trabajador, ambos arreglos ordenados ascendentemente por hora. Uno de ellos contiene los horarios de entrada de los trabajadores, mientras que el otro contiene los de salida.

Una vez cargados los datos en nuestra estructura, se procedió a armar el algoritmo que resuelve el problema. La idea del mismo fue ir indexándo ambos arreglos desde la posición 0 hasta la n-1 ésima. De esta forma, se iba recorriendo el arreglo que contenía los horarios de entrada, hasta que se encontrara una posición en la cuál su horario de entrada fuese mayor al horario de salida que se estaba indexando en ese momento (en el caso de la primer iteración por ejemplo, el horario ubicado en la posición 0 del arreglo de horarios de salida). Mientras se realizaba esto, un contador con valor inicial 0 se iba incrementando con cada iteración, simulando así la cantidad de trabajadores que iban entrando.

Cuando se encontraba un horario de entrada mayor al de salida esto significaba que, antes que ingresara un nuevo trabajador, al menos uno se había retirado, por lo que se salía de esta iteración. En un primer paso se verificaba cuántos trabajadores habían entrado hasta ese momento y se guardaba en una variable (siempre y cuando dicho valor fuese mayor al guardado anteriormente). Luego, se recorría el arreglo que contenía los horarios de salida de una manera parecida a la de antes: recorrer hasta que se encuentre un horario mayor al indexado en el arreglo de entradas, con la diferencia que en este caso, cada vez que se iteraba, se decrementaba el contador de trabajadores simulando un egreso.

Esto se repetía varias veces, hasta que se indexaran todas las posiciones del arreglo de los horarios de entrada, devolviendo finalmente el valor que se iba actualizando tras cada cambio de iteración, es decir, el valor que se actualizaba una vez que se dejaba de iterar el arreglo con los horarios de entrada.

A modo de una explicación más clara que nos acerque un poco más a la implementación, detallamos lo expreso anteriormente a través del siguiente pseudocódigo:

```
Complejidad: O(n)
var i,j,maxJuntos,juntosPorAhora : ℤ
var termine : bool
i ← j ← maxJuntos ← juntosPorAhora ← 0;                              // O(1)
termine ← false;                                                     // O(1)
while (!termine);                                                    // O(n)
do
    while (noLlegueAlFinal(i,n) && hora(entradas[i]) ≤
    hora(salidas[j]));                                               // O(1)
    do
        juntosPorAhora++ ;                                           // O(1)
        i++ ;                                                        // O(1)
    end
    termine = i ≥ n ;                                                // O(1)
    if (maxJuntos ≤ juntosPorAhora);                                 // O(1)
    then
        maxJuntos ← juntosPorAhora;                                  // O(1)
    end
    while (!termine && hora(salidas[j]) ≤ hora(entradas[i]));        // O(1)
    do
        juntosPorAhora−− ;                                           // O(1)
        j++ ;                                                        // O(1)
    end
end
return maxJuntos
```

De esta explicación y de un análisis más minucioso del pseudocódigo, se desprenden algunas hipótesis.

- La complejidad temporal del algoritmo es $O(n)$.
- El peor caso se da cuando hay muchos trabajadores dispersos durante todo el día, es decir, si todos los horarios de salida y entrada entre dos trabajadores cualesquiera son disjuntos.

## 3.3. Análisis de la complejidad del algoritmo

Para realizar el análisis de la complejidad del algoritmo, se decidió utilizar el modelo uniforme y no el logarítmico. Esto se debió a que en este caso, no resulta lógico evaluar la complejidad de acuerdo al costo de representar los valores de los parámetros de entrada. Más aún, por la forma y el contexto del problema y por el algoritmo implementado para la resolución, no sería correcto hacer un análisis logarítmico ya que en este caso las horas representables están acotadas (habíamos dicho que se encontraban entre 00:00:00 y 23:59:59) y la cantidad de trabajadores, aunque no está explícitamente acotada, podríamos suponerla así. De esta manera, sería válido considerar que la complejidad espacial de cada elemento es unitaria, haciendo inadecuado analizar la complejidad del algoritmo con el modelo logarítmico.

Con el objetivo de realizar un mejor análisis de la complejidad del algoritmo propuesto, vamos a analizar el mismo remitiéndonos al pseudocódigo citado en la sección [3.2.1].

Si hacemos una mirada más minuciosa sobre el pseudocódigo, veremos que la complejidad del algoritmo está dada por $n$, que hace referencia a la cantidad de trabajadores de la empresa. Si observamos las complejidades de cada operación, podemos apreciar que todas son constantes, con excepción del flujo “while” más grande (aquél que tiene como condición: ($!termine$) ). Veamos por qué sucede esto.

Como podemos ver en la línea 5, el booleano *termine* es inicializado con el valor de verdad *false*. La otra variable que nos va a interesar para este análisis es $i$, la cuál se inicializa con el valor 0. Con estos valores, y sabiendo que el flujo while que estamos analizando tiene como condición la negación de *termine*, se desprende que el algoritmo terminará sí y solo sí *termine* tome el valor de verdad *true*.

En la línea 13 del pseudocódigo es en el único lugar que se modifica el valor de verdad de *termine*. En esta línea se ve que:

$$termine \leftarrow i \geq n \tag{7}$$

Entonces, juntando esto con lo dicho anteriormente, se puede asegurar que el algoritmo va a finalizar una vez que $i \geq$ n. Veamos entonces cómo se comporta $i$ a lo largo del programa.

La variable $i$ sólo es modificada en el primer while anidado, incrementándose de a uno por vez. Por lo tanto, una vez que se ejecute el contenido de este while anidado $n$ veces, el algoritmo finalizará.

Si consideramos la precondición que dice que todos los trabajadores ingresan a la empresa estrictamente antes de egresar, podemos decir entonces que este primer while anidado va a ejecutarse siempre a lo sumo n veces[7]. Por lo tanto, el algoritmo tiene una complejidad de $O(n)$. Es más, se puede asegurar que el algoritmo nunca va a terminar antes de recorrer toda la lista de ingresos, por lo que el mejor caso es $\Omega(n)$, es decir, el algoritmo es $\Theta(n)$

## 3.4. Detalles de Implementación

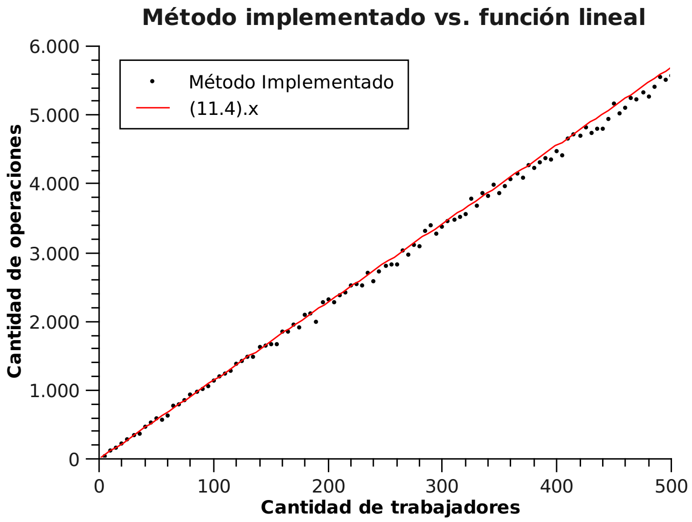
Para compilar este programa, se debe referir a la carpeta que contiene al ejercicio nº 3 y ejecutar make en consola.

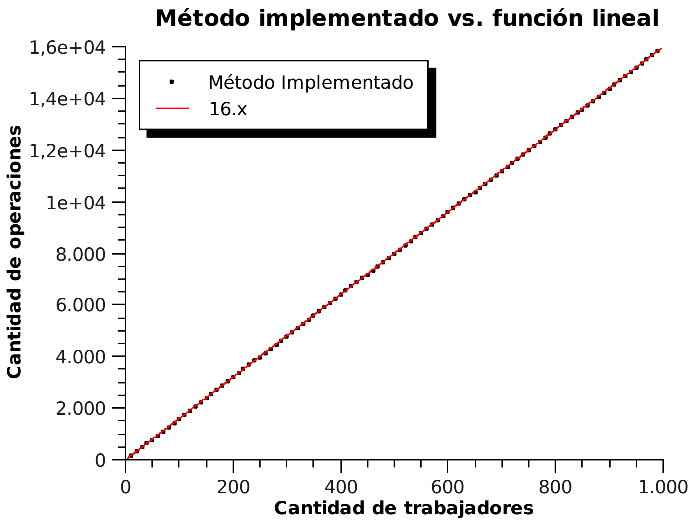
Luego de ejecutar make, se creara un archivo ejecutable, llamado main. Para ejecutarlo, se debe hacer una llamada desde la consola respetando el siguiente formato:
`main [nombreEntrada] [nombreSalida]`.

[7]Ver condición del primer while anidado

Los parámetros son opcionales, pero se puede elegir entre no pasar ningún parámetro, o pasar 2. Si no se pasan parámetros, el programa tomara los valores por defecto, esto quiere decir que el archivo de entrada será el designado por la materia.

## 3.5. Resultados




Cuadro 1: Muestran el comportamiento de la cantidad de operaciones contra la cantidad de trabajadores. El primer gráfico proviene de una entrada hecha al azar. El segundo proviene de evaluar el comportamiento del algoritmo para el peor caso.

## 3.6. Debate

Como podemos observar en los gráficos propuestos en la sección [3.5] en general la implementación realizada se comporta como se esperaba que en teoría lo hiciera. Es decir, durante el desarrollo del informe sobre este ejercicio, entre otras cosas, hicimos notar que nuestra hipótesis sobre la complejidad del algoritmo era que el mismo tenía un costo de $\Theta(n)$, donde $n$ es la cantidad de trabajadores de la empresa.

Se puede identificar en cada gráfico una marcada similitud entre los resultados arrojados en las distintas mediciones que se hicieron sobre la implementación y alguna recta (o función lineal), variando esta última su pendiente para cada caso. Esta pendiente podría compararse o considerarse como la constante que acompaña a la complejidad del algoritmo.

Se puede notar claramente que esta constante o pendiente, es mayor en aquellos gráficos en los cuales se evaluó el comportamiento del algoritmo en el peor caso que en los que se evaluó el mismo con entradas al azar.

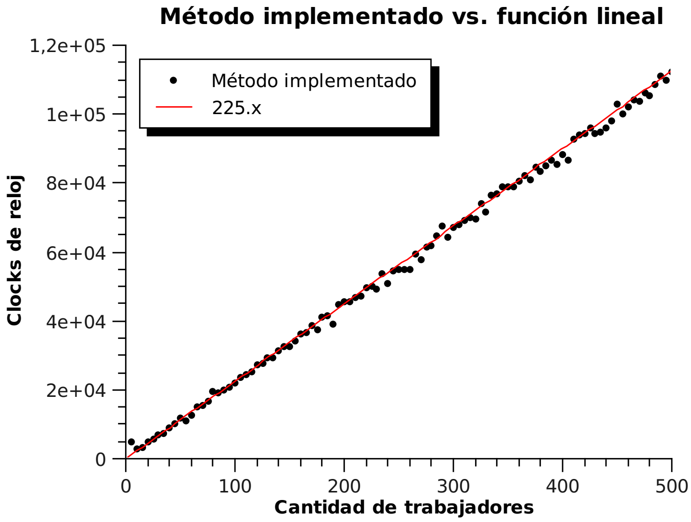
En el segundo gráfico perteneciente al cuadro [2] podemos notar dos casos de outliers. Si observamos más detalladamente se aprecia que estos casos se dan con valores de $n$ grandes. Esto puede deberse a que, como el algoritmo tarda más en terminar[8], es más probable que haya alguna interrupción que irrumpa durante el procesamiento del mismo haciendo que la medición no sea precisa. Esto se debe a la forma en que se miden los ciclos de reloj. Lo que se hace es obtener los ciclos de reloj transcurridos hasta antes de hacer la llamada al algoritmo implementado, luego se toma la misma medición y, finalmente, se toma la diferencia entre ambas obteniendo como resultado la cantidad de ciclos insumidos por el algoritmo. Pero si durante el algoritmo, llega una interrupción y la atención de la rutina de la misma le insume al procesador varias instrucciones, esto puede llegar a provocar una medición de ciclos de reloj muy imprecisa.
Más aún, si tomamos en cuenta cómo un sistema operativo hace manejo de los procesos, puede haber llegado a ocurrir que durante la ejecución del algoritmo, este haya consumido todo su tiempo de ejecución asignado por el sistema, por lo que puede haber tenido que esperar una o incluso varias veces a que el sistema deje continuar con su ejecución.

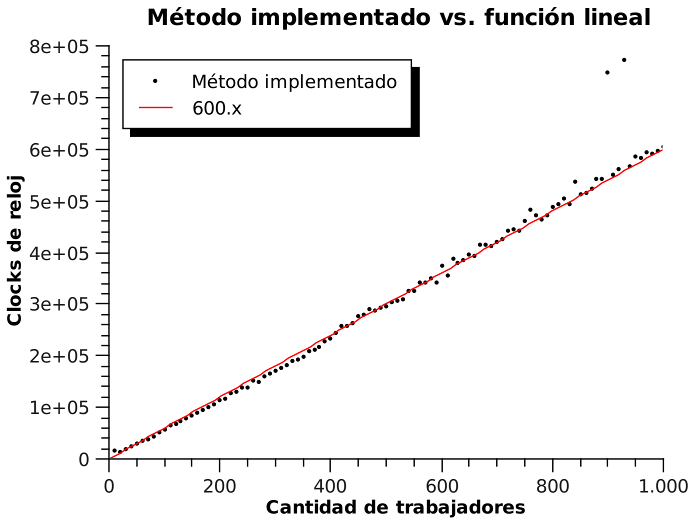
## 3.7. Conclusiones

Luego de describir el funcionamiento del algoritmo, de realizar las pruebas y gráficos pertinentes y de analizarlos detalladamente, podemos realizar algunas conclusiones.
Podemos asegurar fehacientemente que la complejidad del algoritmo propuesto es lineal y que la misma es $\Theta(n)$ donde $n$ es la cantidad de trabajadores. Esto no sólo se desprende del análisis teórico realizado anteriormente, sino también de las sucesivas pruebas realizadas. Claramente se puede observar en todos los gráficos presentados cómo el comportamiento de la implementación se asemeja a una función lineal sobre los datos de entrada.

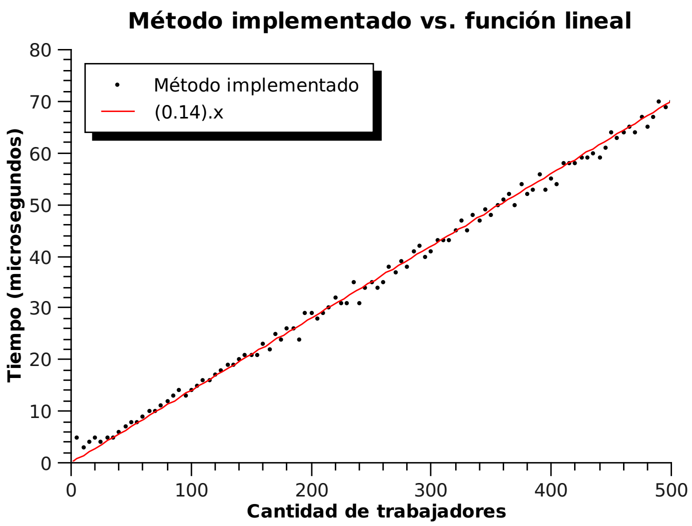
Finalmente, si hacemos una comparación entre los resultados obtenidos al aplicarse el algoritmo sobre datos de entrada azarosos y datos de entrada que se condicen con el peor caso, se puede observar que la constante que acompaña a la complejidad para el peor caso es considerablemente mayor a la constante que aparece en el caso azaroso. Como una conclusión poco menos significativa entonces, se puede decir que los casos en que los horarios de los trabajadores son todos disjuntos son los peores casos para este algoritmo.

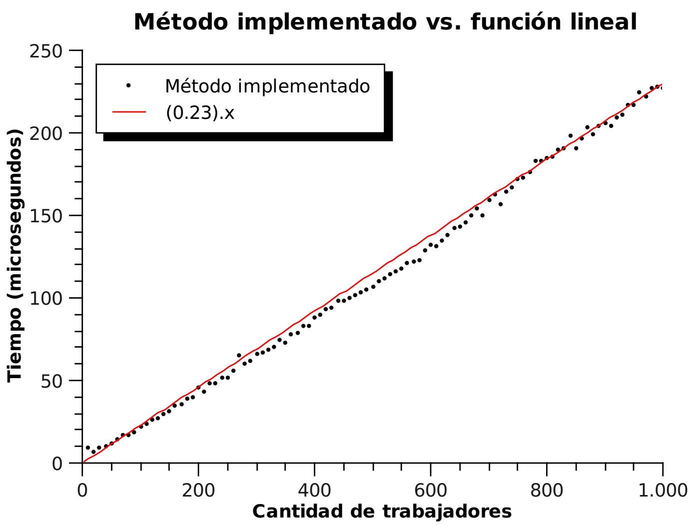
[8] recordar además que este gráfico corresponde al análisis en el peor caso




Cuadro 2: Muestran el comportamiento de la cantidad de ticks de reloj contra la cantidad de trabajadores. El primer gráfico proviene de una entrada hecha al azar. El segundo proviene de evaluar el comportamiento del algoritmo para el peor caso.




Cuadro 3: Muestran el comportamiento de la cantidad de microsegundos contra la cantidad de trabajadores. El primer gráfico proviene de una entrada hecha al azar. El segundo proviene de evaluar el comportamiento del algoritmo para el peor caso.